

Querying Software Abstraction Graphs

Daniel Bildhauer and Jürgen Ebert
University of Koblenz-Landau
{dbildh,ebert}@uni-koblenz.de

Abstract

The analysis of software is supported by query languages, that work directly on the source code or on some of its abstractions. This paper presents the Graph Repository Query Language (GReQL) that works on TGraphs as underlying data structures. The key concepts of the language are described as well as the evaluation and optimization of GReQL queries. The paper concludes with a description of query results and an overview of practical applications of the language.

1 Introduction

Software maintenance and re-engineering is known to be a complex task especially if the source code is the only software artifact available. Query languages allow for uncovering the relevant aspects of the software on the basis of the source code and different kinds of source code abstractions. The **Graph Repository Query Language (GReQL)** is such a query language that works on TGraphs.

TGraphs are typed, attributed, directed and ordered graphs [7]. They are a powerful modeling means. Edges are first class objects, i.e. they have an identity and a type, and they may be attributed. The type system for vertices and edges supports multiple inheritance. Traversal is supported in both directions, and the graphs are subject to algorithms as a whole. Furthermore the ordering of vertices can be used for modeling sequenced occurrences.

2 Source code abstraction

TGraphs are used to make abstractions of source code accessible. Abstract syntax graphs (ASGs) are created out of source code files in a process called fact extraction. Being defined by MOF-compatible meta models [5], these ASGs are a formal and flexible representation which allows for algorithmic as well as query-driven processing. Different fact extractors allow the creation of ASGs for different source languages.

Figure 1 shows a small simplified cutout of the GReQL evaluator source code, which is described later in this paper. The part of the ASG that abstracts this sourcecode is shown in figure 2. While the whole ASG actually contains about 50 vertices and 70 edges, only a small part of it is displayed here.

As vertex *v18* and its edges *e15*, *e17* and *e23* indicate, every variable is modeled by exactly one vertex whereas each of its occurrences in the source code is modeled by a separate corresponding edge. Consequently, the positions of these occurrences in the source code are added as *offset* and *length* attributes to the edges.

```
1 public class DFA extends NFA {
2
3     public DFA(NFA nfa) {
4         eliminateEpsilonTransitions(nfa);
5         myhillConstruction(nfa);
6     }
7
8     void eliminateEpsilonTransitions(NFA nfa) {
9         //code omitted due to space limitations
10    }
11
12    void myhillConstruction(NFA nfa) {
13        //code omitted due to space limitations
14    }
15 }
16
17 public class NFA {
18     //code omitted due to space limitations
19 }
```

Figure 1: Sample Java sourcecode

The types of the vertices and edges in the graph, their incidence structure and the attributes associated with these types and are defined by a *TGraph schema*. Such a schema is a MOF-compatible [13] *metamodel* formulated in the graph modeling language *grUML*, a sublanguage of UML class diagrams.

The practical usage of such TGraphs is supported by the

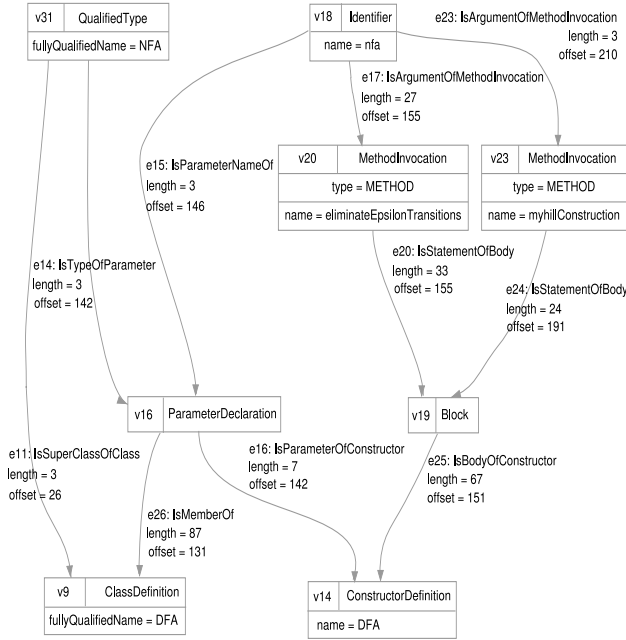


Figure 2: Simplified Graph extracted from code in figure 1

TGraph library JGraLab [11]. It offers features for easy and efficient traversal and manipulation of TGraphs as well as a query interface for the GReQL language. Given a textual GReQL query and a graph, the query interface evaluates the query on the graph to an object of the flexible Java class *JValue*. The elements of the language GReQL are described in the next section. The evaluation of a GReQL query is explained in section 4.

3 Language concepts and elements

The textual syntax of GReQL queries bears some analogy to SQL. One of the main language elements is the *from-with-report* (FWR) query. In the *from*-part of the query, *variables* are bound to *domains*. The variables are then used in the *with*-part to impose *constraints* on their values. The *report*-part defines the structure of the query *result*.

Figure 3 shows a simple *FWR-query* that can be used to find all generalization relationships starting or ending at a class whose name contains the string “Transition”. The variable *e* is bound to all edges of type *IsSuperClassOfClass* in the *from*-clause. The constraint defined in the *with*-clause requires that the *name*-attribute of the start- or endvertex of such an edge contains “NFA” as a substring. The form of the query result is specified by the *report*-clause. For each *IsSuperClassOfClass* edge which satisfies the constraint a row is added to the result table. This table has three columns named *Generalization*, *Superclass*

and *Subclass*, the first column contains the generalization edge while the other two columns contain the names of the classes participating in the generalization. Figure 7 depicts the resulting table.

```

from e: E{IsSuperClassOfClass}
with containsName(startVertex(e).name, "NFA")
      or containsName(endVertex(e).name, "NFA")
report e as "Generalization",
      startVertex(e).name as "Superclass",
      endVertex(e).name as "Subclass" end

```

Figure 3: Query for obtaining all generalization relationships.

Inside GReQL queries, predefined *GReQL functions* can be used, e.g. *containsName*, *startVertex* and *endVertex* in Figure 3. Currently, there exist about 80 predefined functions that reach from mathematical, statistical and string functions to functions on the graph, e.g. *acyclicity* test. Furthermore, also access to the graph schema with all its defined types and restrictions is supported by specific GReQL functions. The function library can be extended by user-defined functions in a flexible way.

Quantified expressions can be used to formulate more sophisticated queries. As each quantified expression is bound only to a finite set of elements, GReQL queries remain algorithmically accessible. An example for a quantified expression in GReQL is shown in figure 4. The query searches for all classes in the abstraction graph which define a parameterless constructor and contain at least one direct subclass that defines a parameterless constructor, too.

```

from super: V{QualifiedType},
      cons: V{ConstructorDefinition}
with super <-- {IsMemberOf} cons
and isEmpty(cons <-- {IsParameterOfConstructor})
and exists sub: super --> {IsSuperClassOfClass},
      subcons: V{ConstructorDefinition}
      @ sub <-- {IsMemberOf} subcons
      and isEmpty(subcons
      <-- {IsParameterOfConstructor})
reportSet super.name
end

```

Figure 4: Query example for quantified expressions

The advantage of graph representations primarily lies in the intuitive representation of relations between entities as edges. Consequently, these relations must be queryable by a graph query language in a convenient and expressive way. In GReQL, *regular path expressions* can be used to formulate queries that utilize the interconnections between entities. Such a regular path expression consists of symbols for directed and typed edges as its basic elements. As an example, the path expression *<-- {IsMemberOf}* in line

three of the above query represents an incoming edge of type `IsMemberOf` or any of its subtypes. Accordingly, `-->` represents an outgoing edge and `<-->` an edge which is either incoming or outgoing. The full expressiveness of these basic path expressions comes with their combination to complex ones. Corresponding to the properties of regular expressions various combinations are possible.

Figure 5 shows a query that contains a complex path expression. It is used to find all class definitions which define a type that may be the type of an argument in a method invocation. Besides simple edge symbols with type restrictions, the path description in lines two to six contains a path alternative, depicted by the symbol `|` and a non-empty path iteration, depicted by `+`. The first edge in the iterated part is optional because it is surrounded by squared brackets.

```

from m:V(MethodInvocation), c:V(ClassDefinition)
with m <--(IsArgumentOfMethodInvocation)
      -->(IsParameterNameOf) <--(IsTypeOfParameter)
      ( <--(IsClassDefinitionOf) |
        ( [->(IsClassDefinitionOf)]
          -->(IsSuperclassOfClass) )+ ) c
report m as "MethodInvocation",
        c as "ClassDefinition" end

```

Figure 5: Query for obtaining class definitions of all types that can be used as argument types in a method invocation.

Appart from edge descriptions also more complex constructs are possible as basic path expression elements. As an example, `-e-> v -->` claims the edge `e` in the outgoing direction followed by the vertex `v` and any other edge.

Besides the checking for *reachability* between vertices and finding sets of reachable nodes as shown in the above queries, regular path expressions can also be used to calculate *paths* and *systems of paths* in graphs. An example for such a path system creation is shown in the query in figure 6. The query finds all direct and indirect calls of methods that are marked as deprecated. For each deprecated method, a set of paths from its callers is created by an application of the GReQL function `pathSystem`.

```

from method:V(Method)
with contains(method.annotation, "deprecated")
report pathSystem(method, <--(Calls)+ &(Method))
end

```

Figure 6: Query for obtaining calls of deprecated methods

4 Evaluating GReQL queries

The evaluation of textual GReQL queries as shown in the previous section starts with *parsing* them into a directed

acyclic syntax TGraph. During the parsing process, some simplifications are made, e.g. variables are represented by only one vertex for each variable and occurrences of variables are modeled as edges (cf. Figure 1)

To make GReQL usable for interactive work, various *optimizations* are performed on the query before it is eventually evaluated [10]. All these optimizations are carried out directly on the query TGraph. Query optimization is well-explored in relational systems (see [3] as an overview). Consequently, these optimization techniques are adapted and used in GReQL wherever possible. The most important optimizations are: selection as early as possible, unification of common subgraphs and reordering of variables and predicates.

After the optimization phase, the query graph is *evaluated* in a *syntax-driven* manner. The results of the vertices in the GReQL syntax graph are synthesized from the results of their child vertices [2].

The *evaluation of path descriptions* is done by an automaton-driven graph traversal. Using Thompson's construction [14], the regular path expressions are transformed into a nondeterministic finite automaton (NFA). To improve evaluation efficiency, epsilon-transitions are eliminated and the NFAs are transformed into deterministic finite automata (DFA) using the Myhill's construction algorithm [9]. In the worst case, a DFA that accepts exactly the same language as a NFA with n states could have up to 2^n states, but this explosion does not occur practically.

To perform the path search, the DFAs are used to drive a worklist algorithm that traverses the graph and marks the vertices with the states of the DFA. This marking ensures that the search algorithms terminates and that its complexity is $O(|S| \cdot \max(|V|, |E|))$ where S is the state set of the DFA and V and E are the vertex and edge set of the graph, respectively.

5 Query results

The *result* of an evaluation may be an arbitrarily complex data structure which is realized by the facade class `JValue`. Simple data types like integers or single vertices or edges as well as more complex ones such as sets, bags, lists or tables are stored in a common data structure. Furthermore, even graph-like data structures like subgraphs, paths and path systems may be part of a query result. Every element in the result has an reference to the vertex or edge it is calculated from, e.g. an integer value representing the degree of a vertex has a link to that vertex.

To benefit as much as possible from such a result, it can be accessed in several ways. It can either be processed algorithmically using Java, or it can be exported to various display formats, e.g. XML or HTML. Figure 7 shows an HTML-view of the result of the query in figure 3. All un-

Graph id:	e5270a06-7a3bd33-2b55f2db-3831609a
Result size:	20

Generalization	Superclass	Subclass
e1: IsSuperClassOfClass	v31: QualifiedType	v9: ClassDefinition

Figure 7: Result of query in figure 3

derlined parts are hyperlinks to the elements in the graph where the appropriate information originates from.

6 Practical usage and related work

GReQL is in practical usage since the mid of the 1990s in various projects. Its first version was developed for the reverse engineering tool GUPRO¹ (Generic Understanding of Programs, [8]). GUPRO is a tool environment for software analysis. Based on precisely defined metamodels, an abstraction graph is generated out of the source code of the program to analyze. GReQL queries are then used to find information in the abstraction graph as shown above.

In the EU-STREP project ReDSeeDS², which intends to support reuse of software projects on the basis of their requirements, GReQL is used to determine possible reuse candidates. Abstractions of software projects are stored as TGraphs and GReQL is used to perform queries on these abstraction graphs [4, 6].

Apart from the XQuery and XPath XML query languages and SPARQL as a language to query RDF graphs in the context of the semantic web, there are some languages more related to GReQL that work on graph models directly. One such language is Gram [1] which includes walks and hyperwalks as a concept similar to paths and path systems. Another one is G+ [12] which supports regular path expressions and an evaluation by an automaton driven search similar to the one used in GReQL. GReQL combines these features with the full power of TGraphs.

7 Conclusion

This paper described the query language GReQL and its application in source code analysis. The general approach of extracting information from source code into abstract syntax graphs as a flexible data structure is a basis for efficient and versatile analysis. The graph category of TGraphs and the approach of modeling identifiers and variables by single vertices and their different occurrences by dedicated edges leads to a compact representation with redundancies.

A coarse overview of the syntax and semantics of GReQL was given using some sample queries together with

a more detailed excursion into regular path expressions as the mechanism to query relationships in graphs. The evaluation and optimization process including the evaluation of regular path expressions using automata driven graph traversals was sketched shortly.

References

- [1] B. Amann and M. Scholl. Gram: A graph data model and query language. In *European Conference on Hypertext*, 1992.
- [2] D. Bildhauer. *Auswertung der TGraphanfragesprache GReQL 2*. VDM Verlag Dr. Müller, Saarbrücken, 2008.
- [3] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, New York, USA, 1998. ACM.
- [4] J. Ebert. Metamodel-based Querying of Software Artifacts. In *Proceedings of the International Workshop on Model Reuse Strategies (MoRSe 2006)*, 10 2006.
- [5] J. Ebert. Metamodels Taken Seriously: The TGraph Approach. In K. Kontogiannis, C. Tortjjs, and A. Winter, editors, *12th European Conference on Software Maintenance and Reengineering*, Piscataway, NJ, 2008. IEEE Computer Society.
- [6] J. Ebert, D. Bildhauer, V. Riediger, and H. Schwarz. Using the TGraph Approach for Model Fact Repositories. In *Proceedings of the International Workshop on Model Reuse Strategies (MoRSe 2008)*, 5 2008.
- [7] J. Ebert and A. Franzke. A declarative approach to graph based modeling. In *Graph-Theoretic Concepts in Computer Science*, volume 903/1995 of *Lecture Notes in Computer Science*, pages 38–50. Springer Berlin / Heidelberg, 1995.
- [8] J. Ebert, B. Kullbach, V. Riediger, and A. Winter. GUPRO. Generic Understanding of Programs - An Overview. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.
- [9] J. Hopcroft and J. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Addison-Wesley, Reading, Massachusetts, USA, 1990.
- [10] T. Horn. Ein Optimierer für GReQL 2. Diplomarbeit, University of Koblenz-Landau, Institute for Software Technology, 2008.
- [11] S. Kahle. JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen. Diplomarbeit, University of Koblenz-Landau, Institute for Software Technology, 2006.
- [12] A. Mendelzon and P. Wood. Finding regular simple paths in graph databases. In *Proceedings of the Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA) 15, Amsterdam.*, August 1989.
- [13] Object Management Group. *Meta Object Facility (MOF) Core Specification, OMG Available Specification, Version 2.0, formal/2006-01-01*, 2006.
- [14] K. Thompson. Regular expression search algorithms. *Communications of the ACM*, (11):419–422, 6 1968.

¹www.gupro.de

²www.redseeds.eu